Protocolo de Testes de Segurança para Servidor Flask

1. Planejamento e Preparação

- Ambiente de Testes Isolado: os testes foram realizados em um ambiente isolado, com um servidor de desenvolvimento.

- Ferramentas Utilizadas:
  - Burp Suite, OWASP ZAP, Postman, sqlmap, etc., para automação de testes de segurança.
  - Flask-Security, Flask-Login, ou outras bibliotecas para autenticação/segurança.
  - Testes de carga com ferramentas como Apache JMeter ou Locust.

2. Testes de Segurança

a) SQL Injection

Objetivo: Garantir que não haja vulnerabilidades de SQL Injection, onde um atacante pode manipular suas consultas.

Teste Manual:

- Enviar entradas maliciosas como "' OR 1=1 --", "' DROP TABLE pessoas --", etc., no campo de pesquisa.
- Verificar se a aplicação responde de maneira segura e se o banco de dados não é afetado.

Teste Automatizado:

- Usar ferramentas como sqlmap para testar automaticamente a aplicação em busca de vulnerabilidades de SQL Injection.

  Exemplo:

  sqlmap -u "http://127.0.0.1:5000/consulta" --data="termo=' OR 1=1 --" --batch

b) Cross-Site Scripting (XSS)

Objetivo: Garantir que a aplicação não permita a execução de scripts maliciosos no navegador dos usuários.

Teste Manual:
- Enviar entradas com scripts, como "<script>alert('XSS')</script>", em campos de entrada.
- Verificar se o conteúdo é escapado corretamente na renderização da página ou se há algum erro de execução.

Teste Automatizado:
- Usar ferramentas como OWASP ZAP para realizar testes automáticos de XSS.

c) Cross-Site Request Forgery (CSRF)

Objetivo: Verificar se a aplicação está protegida contra CSRF, onde um atacante pode induzir um usuário a executar ações indesejadas.

Teste Manual:
- Fazer uma requisição de formulário sem um token CSRF ou com um token forjado e ver se a requisição é aceita.
- Verificar se a aplicação retorna um erro adequado de CSRF (ex: "Token CSRF inválido").

Teste Automatizado:
- Usar Flask-WTF ou similar para garantir que a proteção CSRF esteja configurada corretamente.
- Para testar, usar ferramentas como Burp Suite ou Postman para tentar enviar requisições sem o token CSRF.

d) Autenticação e Autorização

Objetivo: Garantir que apenas usuários autenticados e autorizados possam acessar recursos sensíveis.

Teste Manual:
- Tentar acessar áreas restritas sem estar logado.
- Tentar acessar dados de outro usuário, manipulando os parâmetros de URL ou fazendo requisições com privilégios elevados.

Teste Automatizado:
- Usar ferramentas como Flask-Login e Flask-Security para validar se a autenticação e a autorização estão funcionando corretamente.
- Usar ferramentas de automação de login para verificar as falhas de autenticação.

e) Clickjacking

Objetivo: Verificar se a aplicação é vulnerável a ataques de Clickjacking, onde um iframe malicioso pode induzir o usuário a clicar em elementos invisíveis.

Teste Manual:
- Tentar embutir a página da aplicação em um iframe usando o código HTML:
  <iframe src="http://127.0.0.1:5000" style="height: 100%; width: 100%;"></iframe>
- Se o conteúdo for carregado sem bloqueio, a aplicação é vulnerável a Clickjacking.

Teste Automatizado:
- Verificar se a política de cabeçalho X-Frame-Options está configurada corretamente para evitar o carregamento em iframes.

3. Testes de Performance e Robustez

a) Testes de Carga (Load Testing)

Objetivo: Garantir que o servidor consegue lidar com uma alta quantidade de requisições simultâneas sem falhas.

Ferramentas:
- Apache JMeter ou Locust para testar a escalabilidade e a capacidade de resposta do servidor.
- Configurar para enviar um número crescente de requisições e verificar se o servidor continua a responder adequadamente sem travar ou cair.

b) Testes de Resiliência (Stress Testing)

Objetivo: Testar como o servidor se comporta em situações extremas, como altas cargas ou condições inesperadas.

Ferramentas:
- Uso de Locust para gerar tráfego de teste e simular o comportamento de múltiplos usuários simultâneos.
- Testar duração prolongada de tráfego ou altas taxas de erro.

4. Testes de Proteção de Dados Sensíveis

a) Proteção de Senhas

Objetivo: Verificar se as senhas dos usuários estão sendo armazenadas de forma segura.

Teste Manual:
- Verificar no banco de dados se as senhas são criptografadas ou hashadas (por exemplo, usando bcrypt ou argon2).

b) Proteção de Sessões

Objetivo: Verificar a segurança das sessões de usuários e se os cookies estão configurados corretamente.

Teste Manual:
- Verificar os cookies da aplicação, e assegurar-se de que os cookies de sessão estão configurados com as flags `HttpOnly`, `Secure` e `SameSite`.

5. Testes de Arquitetura e Configuração

a) Segurança de Configuração

Objetivo: Validar se o Flask está sendo executado em modo de produção e se as configurações de segurança estão ativas.

Teste Manual:
- Configuração do Flask, assegurando de que debug=False e SECRET_KEY está definido.

b) Revisão de Código

Objetivo: Realizar uma revisão do código para garantir que não existam práticas inseguras ou código desnecessário.

Checklist:
- SQL parametrizado.
- Validação e escape de entradas do usuário.
- Uso de cabeçalhos de segurança adequados (X-Content-Type-Options, X-Frame-Options, etc.).
- Proteção contra ataques de força bruta e rate limiting.

6. Testes de Logs e Auditoria

a) Auditoria e Logs

Objetivo: Verificar se o sistema registra adequadamente as ações críticas para a segurança, como login, falhas, inserção de dados, etc.

Teste Manual:
- Tentar realizar ações críticas (login, mudanças de dados) e verificação de registros nos logs.

b) Testes de Monitoramento

Objetivo: Validar que o servidor está protegido contra ataques em tempo real, com ferramentas adequadas de monitoramento.

7. Testes de Correção e Regressão

Objetivo: Verificar se as correções e melhorias aplicadas para corrigir vulnerabilidades anteriores não introduziram novos problemas.

- Realização de testes de regressão para garantir que mudanças não impactem funcionalidades existentes.

Conclusão

Este protocolo de testes visa cobrir as principais áreas de segurança e resiliência de um servidor Flask, garantindo que ele seja resistente a diversos tipos de ataques.

É sempre importante lembrar que a segurança é um processo contínuo.
Portanto, após a correção das vulnerabilidades, é dever do desenvolvedor continuar monitorando e realizando testes periódicos.